UNIVERSIDADE FEDERAL DE LAVRAS

TRABALHO 1
Algoritmo e Estrutura de Dados III - GCC109

Alunos: Augusto Soares Pereira - 201210320
Ueslei Marcelino da Guia - 201120619

Lavras - MG
2014

1. Introdução

O objeto do trabalho é implementar de diversas maneiras, algoritmos que iram contar palavras, tendo como base os conceitos de Árvore Binária de Busca, Árvore AVL e Árvore DSW.

Este trabalho apresenta 4 alternativas de aceleração, ou seja, métodos que iram contar quantas palavras apareceram mais vezes nos arquivos de textos. Ao decorrer do mesmo, será explicada de forma detalha cada método.

2. Árvore Binária de Busca

Esta árvore é uma estrutura de dados de árvore binária baseada em nós, onde todos os nós da sub-árvore esquerda possuem um valor numérico inferior ao nó raiz, e todos os nós da sub-árvore direita possuem valor superior ao nó raiz. O objetivo desta árvore é estruturar os dados de forma flexível, permitindo busca binária. Operações em uma árvore binária requerem comparações entre nós. Essas comparações são feitas com chamadas a um comparador, que é uma sub-rotina que calcula a ordem total em dois valores quaisquer. Esse comparador pode ser explícita ou implicitamente definido, dependendo da linguagem em que a árvore binária de busca está implementada.

A função `Insere_ou_Processa(struct No** R, Item I)` presente na implementação refente a árvore binária é a mais simples de todas. A sua função resume apenas em inserir a próxima palavra na árvore a direita se a sua chave for maior que a chave que está sendo comparada ou a esquerda se for menor.

A sua implementação é a seguinte:

```
void Insere_ou_Processa(struct No** R, Item I)
{
     struct No *r = *R;

     //Encontrou o ponto de inserção
     if(r == NULL)
     {
          I->contador = 1;
          *R = Novo(I, NULL, NULL);
          return;
     }

     //Inserção à esquerda
     if( Menor( Get_Chave( I ), Get_Chave( r->I ) ) )
     {
          Insere_ou_Processa( &(r->esq), I );
          return;
     }

     //Inserção à direita
     if( Menor( Get_Chave( r->I ), Get_Chave( I ) ) )
     {
          Insere_ou_Processa( &(r->dir), I );
```

```
        return;
    }

    Incrementa_contador((*R)->I);
}
```

3. Árvore AVL

Esta é uma árvore de busca binária auto-balanceada. Em tal árvore, as alturas das duas sub-árvores a partir de cada nó diferem no máximo em uma unidade. As operações de busca, inserção e remoção de elementos possuem complexidade O(log 2n), no qual n é o número de elementos da árvore. Inserções e remoções podem também requerer o rebalanceamento da árvore, exigindo uma ou mais rotações.

A função `Insere_ou_Processa(struct No** R, Item I)` na árvore AVL já é um pouco mais complexa. É necessário fazer o balanceamento a cada inserção na árvore por meio de rotações simples e duplas tanto a direita como a esquerda.

A sua implementação será da seguinte forma:

```
void Insere_ou_Processa(struct No** R, Item I)
{
    struct No *r = *R;

    //Encontrou o ponto de inserção
    if(r == NULL)
    {
        I->contador = 1;
        *R = Novo(I, NULL, NULL);
        return;
    }

    //Inserção à esquerda
    if( Menor( Get_Chave( I ), Get_Chave( r->I ) ) )
    {
        Insere_ou_Processa( &(r->esq), I );
        DIC_Balanceia_AVL(R);
        return;
    }

    //Inserção à direita
    if( Menor( Get_Chave( r->I ), Get_Chave( I ) ) )
    {
        Insere_ou_Processa( &(r->dir), I );
        DIC_Balanceia_AVL(R);
        return;
    }

    Incrementa_contador((*R)->I);
```

```
}
```

A função responsável pelo balanceamento é a `DIC_Balanceia_AVL`, que é chamada duas vezes. Ela é chamada na inserção a direita e a esquerda.

4. Estratégia de Aceleração 1 – EA1

A função `Insere_ou_Processa(struct No** R, Item I)` presente na implementação EA1 recebe, como nas outras estratégias, um ponteiro de ponteiro para um nó e também o item que se deseja inserir na árvore.

A EA1 consiste em levar para cima o nó, toda vez que a chave é acessada. Este teste é realizado tando na inserção a esquerda quanto na inserção a direta, por meio das funções `DIC_Rotaciona_Esquerda` e `DIC_Rotaciona_Direita` respectivamente.

```
int Insere_ou_Processa_EA1(struct No** R, Item I)
{
     struct No *r = *R;

     //Encontrou o ponto de inserção
     if(r == NULL)
     {
          I->contador = 1;
          *R = Novo(I, NULL, NULL);
          return 1;
     }

     //Inserção à esquerda
     if( Menor( Get_Chave( I ), Get_Chave( r->I ) ) )
     {
          int a = Insere_ou_Processa_EA1( &(r->esq), I );
          if (a == 1)   DIC_Rotaciona_Direita(R);
          return 0;
     }

     //Inserção à direita
     if( Menor( Get_Chave( r->I ), Get_Chave( I ) ) )
     {
          int a = Insere_ou_Processa_EA1( &(r->dir), I );
          if (a == 1) DIC_Rotaciona_Esquerda(R);
          return 0;
     }

     Incrementa_contador((*R)->I);
     return 1;
}
```

5. Estratégia de Aceleração 2 – EA2

A EA2 baseia-se em múltiplas rotações. Toda vez que é inserido uma nova chave, o seu nó será rotacionado até a raiz da árvore.

```
void Insere_ou_Processa_EA2(struct No** R, Item I)
{
     struct No *r = *R;

     //Encontrou o ponto de inserção
     if(r == NULL)
     {
          I->contador = 1;
          *R = Novo(I, NULL, NULL);
          return;
     }

     //Inserção à esquerda
     if( Menor( Get_Chave( I ), Get_Chave( r->I ) ) )
     {
          Insere_ou_Processa_EA2( &(r->esq), I );
          DIC_Rotaciona_Direita(R);
          return;
     }

     //Inserção à direita
     if( Menor( Get_Chave( r->I ), Get_Chave( I ) ) )
     {
          Insere_ou_Processa_EA2( &(r->dir), I );
          DIC_Rotaciona_Esquerda(R);
          return;
     }

     Incrementa_contador((*R)->I);
}
```

6. Estratégia de Aceleração 3 – EA3

A estratégia de aceleração 3 apresenta um estrutura a mais que as outras duas estratégia. Então na função `Insere_ou_Processa(struct No** R, Item I)` é necessário chamar a função responsável em "armazenar" os dados no arranjo.

A `Insere_ou_Processa(struct No** R, Item I)` foi implementada da seguinte forma:

```
void EA3_Insere_ou_Processa(struct No** R, Item I)
{
```

```
    struct No *r = *R;

    //Encontrou o ponto de inserção
    if(r == NULL)
    {
        I->contador = 1;
        *R = Novo(I, NULL, NULL);
        EA3_Insere_B(*R);
        return;
    }

    //Inserção à esquerda
    if( Menor( Get_Chave( I ), Get_Chave( r->I ) ) )
    {
        EA3_Insere_ou_Processa( &(r->esq), I );
        return;
    }

    //Inserção à direita
    if( Menor( Get_Chave( r->I ), Get_Chave( I ) ) )
    {
        EA3_Insere_ou_Processa( &(r->dir), I );
        return;
    }

    Incrementa_contador(r->I);
    EA3_Insere_B(r);
}
```

E a função que insere no arranjo auxiliar recebe um valor do tipo pNo, ponteiro para um nó. A sua implementação ficou da seguinte forma:

```
void EA3_Insere_B(pNo no){

    int i;
    for (i = TAM_B - 1; i > 0; i--){
        B[i] = B[i - 1];
    }
    B[i] = no;
}
```

7. Estratégia de Aceleração 4 – EA4

A estratégia de aceleração 4 foi inspirada no algoritmo de substituição de página Segunda Chance utilizado pelos Sistemas Operacionais quando se deseja substituir uma página que não seja intensamente usada, por outra.

Foi necessário criar uma estrutura auxiliar que irá armazenar todos os ponteiros responsáveis em apontar todas as 10 palavras que mais apareceram no texto.

A função `Insere_ou_Processa(struct No** R, Item I)` ficou da seguinte forma, bastante parecida com a EA3:

```
void EA4_Insere_ou_Processa(struct No** R, Item I)
{
     struct No *r = *R;

     //Encontrou o ponto de inserção
     if(r == NULL)
     {
          I->contador = 1;
          *R = Novo(I, NULL, NULL);
          EA4_Insere_B(*R);
          return;
     }

     //Inserção à esquerda
     if( Menor( Get_Chave( I ), Get_Chave( r->I ) ) )
     {
          EA4_Insere_ou_Processa( &(r->esq), I );
          return;
     }

     //Inserção à direita
     if( Menor( Get_Chave( r->I ), Get_Chave( I ) ) )
     {
          EA4_Insere_ou_Processa( &(r->dir), I );
          return;
     }

     Incrementa_contador(r->I);
     EA4_Insere_B(r);
}
```

A função responsável por inserir o elemento na estrutura auxiliar é a seguinte:

```
void EA4_Insere_B(pNo no){

     int i;
     while (B[TAM_B - 1].segundaChace == 1)
     {
          EA4_Segunda_Chance();
     }
```

```
     for (i = TAM_B - 1; i > 0; i--){
          B[i].no = B[i - 1].no;
          B[i].segundaChace = B[i - 1].segundaChace;
     }

     B[i].no = no;
     B[i].segundaChace = 1;
}
```

E aqui a função responsável por realizar a análise de cada chave, dando ou não uma segunda chance para a mesma.

```
void EA4_Segunda_Chance()
{
     NO_B aux;
     int i;

     aux.no = B[TAM_B - 1].no;

     for (i = TAM_B - 1; i > 0; i--){
          B[i].no = B[i - 1].no;
          B[i].segundaChace = B[i - 1].segundaChace;
     }

     B[i].no = aux.no;
     B[i].segundaChace = 0;
}
```

8. Resultados

Os seguintes resultados foram obtidos através do terminal do Ubuntu 12.04 com o auxilio do comando `time`.

Implementação	T1.txt	T2.txt	T3.txt
ABB	4.376s	0.709s	0.120s
AVL	5.265s	0.956s	0.172s
EA1	4.437s	0.883s	0.148s
EA2	4.495s	0.938s	0.130s
EA3	6.806s	2.110s	0.242s
EA4	8.425s	2.937s	0.324s

Tabela 1: Tempos de Execução

9. Discussão

Após os testes realizados com os algoritmos, o tempo dos resultados obtidos foi calculada, e o rendimento dos mesmos avaliados pela dupla. A partir dessas avaliações podemos classificar os dados resultados para servir como base para a avaliação da melhor e o pior estratégia de aceleração. Os resultados estão listados logo abaixo:

1 – O algoritmo da árvore ABB apresentou os melhores resultados em todos os textos utilizados para o teste.

2 – Embora tenha obtido resultados parecidos com a Árvore ABB, a Árvore AVL demonstrou menor eficiência em relação ao tempo de inserção, devido ao tempo dedicado aos balanceamentos.

3 – A estratégia de aceleração EA1 apresentou o melhor resultado no texto 2 em comparação com as outras estratégia de aceleração, porém o resultado com o texto 3 foi pior que as outras estratégia. Podemos afirmar então, que o algoritmo se comporta melhor com textos maiores.

4 – As duas últimas estratégias apresentação os piores resultados em todos os textos utilizados para a realização dos testes. Porém, acreditamos que a estratégia de aceleração EA4 seja a melhor opção quando se quiser saber quais são as 10 palavras que mais apareceram. A busca pelas palavras é feita utilizando a estrutura auxiliar, e não é necessário realizar a busca diretamente na árvore.

10. Compilação dos Arquivos

A compilação dos arquivos é feita através do arquivo Makefile da seguinte forma:

```
make contadordepalavrasABB
./ contadordepalavrasABB <T1.txt

make contadordepalavrasAVL
./ contadordepalavrasAVL <T1.txt

make contadordepalavrasEA1
./ contadordepalavrasEA1 <T1.txt

make contadordepalavrasEA2
./ contadordepalavrasEA2 <T1.txt

make contadordepalavrasEA3
./ contadordepalavrasEA3 <T1.txt

make contadordepalavrasEA4
./ contadordepalavrasEA4 <T1.txt
```